~~PROTOCOL FOR MANAGING AND~~
A METHOD ~~OF~~FOR TRANSFORMING AND VERIFYING ~~AND OF CONVERTING A~~ DOWNLOADED PROGRAM ~~FRAGMENT,~~FRAGMENTS WITH DATA TYPES RESTRICTIONS AND
CORRESPONDING ~~SYSTEMS~~SYSTEM

BACKGROUND OF THE INVENTION

1. Field of the invention

The invention relates to a ~~protocol~~process for managing, a method of verifying and a method of transforming a downloaded program fragment and the corresponding systems, more particularly intended for ~~on-board~~embedded data-processing systems ~~having few resources available in terms of~~with limited memory and of computing power resources.

2. Prior Art

In a general way, by reference to ~~figure~~Figure 1a, it is reiterated that ~~on-board~~embedded data-processing systems 10 include a microprocessor 11, a permanent memory, such as a non-writable memory 12 containing the code of the executable program, and a rewritable, nonvolatile, permanent memory ~~15~~13 of EEPROM type containing the data stored in the system, a volatile, random-access memory 14 in which the program stores its intermediate results while it is executing, and input/output devices 15 allowing the system to interact with its environment. ~~In~~ When the ~~case in which the on-board~~embedded data-processing system consists of a microprocessor card, of the bank-card type, the input/output device 15 consists of a serial link allowing the card to communicate with a terminal, such as a card-reader terminal.

In conventional ~~on-board~~embedded data-processing systems, the code of the program executed by the system is fixed during construction of the system, or, at the latest, when the latter is customized before delivery to the final user.

More sophisticated ~~on-board~~embedded data-processing systems have, however, been implemented, these systems being reprogrammable, such as microprocessor cards of the ~~JavaCard~~JAVACARD type, for example. ~~With respect~~ Compared to the ~~preceding~~earlier types, these reprogrammable systems add the possibility of enhancing the program after the system has been put into service, via an operation of downloading program fragments. These program fragments ~~of programs, widely designated by~~, commonly known as "applets", will be ~~designated~~referred to as applets or program fragments indiscriminately in the present description. For a more detailed description of ~~JavaCard~~JAVACARD systems, reference could usefully be made to the documentation published by the company SUN MICROSYSTEMS INC., and in particular to the electronically available documentation, ~~JavaCard~~JAVACARD technology chapter, on the ~~www (~~World Wide Web~~)~~ at site http://java.sun.com/products/javacard/index.html, available since June 1999.

~~figure~~Figure 1b illustrates the architecture of such a reprogrammable ~~on-board~~embedded data-processing system. This architecture is similar to that of a conventional ~~on-board~~embedded system, ~~with~~apart from the ~~difference~~fact that the reprogrammable ~~on-board~~embedded system can ~~moreover~~in addition receive applets ~~by way of~~via one of its input/output devices, then store them in its permanent memory 13 from which they can then be executed ~~as a supplement to~~complementing the main program.

For reasons of portability between different ~~on-board~~embedded data-processing systems, applets ~~are presented in~~take the form of code for a standard virtual machine. This code is not directly executable by the microprocessor 11, but it has to be interpreted in software terms by a virtual machine 16, which consists of a program resident in a non-writable permanent memory 12. In the abovementioned example of ~~JavaCard~~JAVACARD cards, the virtual machine used is

a subset of the ~~Java~~JAVA virtual machine. For a description of the specifications relating to the ~~Java~~JAVA virtual machine and of the virtual machine used, reference could usefully be made to the work published by Tim LINDHOLM and Frank YELLIN, entitled "The Java Virtual Machine Specification~~;~~", Addison-Wesley 1996, and to the documentation published by the company SUN MICROSYSTEMS INC. "~~JavaCard~~JAVACARD 2.1 Virtual Machine Specification", documentation available electronically on the ~~www~~World Wide Web at site http://java.sun.com/products/javacard/JCVMSpec.pdf, since March 1999.

The operation of downloading applets onto an ~~on-board~~embedded data-processing system in service poses considerable security problems. An applet which is ~~involuntarily~~unintentionally, or even deliberately, badly written may incorrectly modify data present on the system, prevent the main program from ~~executing~~being executed correctly or at the right time, or ~~else~~even modify other applets downloaded previously, making them unusable or harmful.

An applet written by a computer hacker may also divulge confidential information stored in the system, information such as the access code in the case of a bank card, for example.

At the present time, three solutions have been proposed with a view to remedying the problem of applet security ~~of applets~~.

A first solution consists in using cryptographic signatures~~, so as~~ in order to accept only applets originating from trusted ~~bodies~~organizations or persons.

In the abovementioned example of a bank card, only the applets bearing the cryptographic signature of the bank having issued the card are accepted and executed by the card, any other unsigned applet being rejected in the course of the downloading operation. An ill-intentioned user of the card, not having available encryption keys from the bank, will therefore be ~~incapable of executing~~unable to execute an unsigned and dangerous applet on the card.

This first solution is well ~~adapted~~suited to the case where all the applets originate from the same single source, the bank in the abovementioned example. This solution is difficult to apply in the case ~~in which~~where the applets originate from several sources, such as, in the example of a bank card, the card manufacturer ~~of the card~~, the bank, the ~~bodies~~organizations managing ~~services by~~ bank card services, the large commercial distribution organizations offering ~~clientele~~customers loyalty programs and ~~proposing,~~ legitimately~~,~~ offering to download specific applets onto the card. The sharing and the holding among these various economic participants of the encryption keys necessary for the electronic signature of the applets pose major technical, economic and legal problems.

A second solution consists in carrying out dynamic ~~checks on~~ access and ~~on~~ typing ~~during the execution of~~checks while executing the applets.

In this solution, the virtual machine carries out a certain number of checks, ~~during the execution of~~while executing the applets, such as:

- ~~check of~~memory access ~~to the memory~~check: upon each read or write in a memory area, the virtual machine verifies the right of access by the applet to the corresponding data;
- dynamic verification of the data types: upon each instruction from the applet, the virtual machine verifies that the constraints on the data types are satisfied. By way of example, the virtual machine may ~~have~~apply special ~~handling for~~processing to data such as valid memory addresses, and prevent the applet generating invalid memory addresses by way of integer/address conversions or arithmetic operations on the addresses;
- detection of stack overflows and of illegal accesses to the execution stack of the virtual machine, which, under certain conditions, are likely to disturb the operation thereof, to the point of circumventing the preceding check mechanisms.

This second solution allows execution of a wide range of applets under satisfactory security conditions. However, it ~~features~~presents the drawback of a considerable slowing of the execution, caused by the range of dynamic verifications. In order to obtain a reduction in this slowing effect, some of these verifications can be ~~taken charge of~~managed by the microprocessor itself, at the cost, however, of an increase in the complexity thereof and thus of the cost price of the ~~on-board~~embedded system. Such verifications furthermore increase the ~~requirements for~~ random-access and permanent memory requirements of the system, ~~by reason~~because of the additional type information which ~~it is necessary to associate~~must be associated with the data handled.

A third solution consists in carrying out a static verification of the code of the applet during the downloading.

In this solution, this static verification simulates the execution of the applet at the level of the data types and establishes, once and for all, that the code of the applet complies with the rule of data types and of access control imposed by the virtual machine and does not cause a stack overflow. If this static verification is successful, the applet can then be executed without it being necessary dynamically to verify that this rule is complied with. In the event that the static verification process fails, the ~~on-board~~embedded system rejects the "applet" and does not allow its subsequent execution. For a more detailed description of the abovementioned third solution, reference could usefully be made to the work published by Tim LINDHOLM and Frank YELLIN quoted above, to the article published by James A. GOSLING entitled "Java Intermediate Byte Codes", proceedings of the ACM SIGPLAN, Workshop on Intermediate Representations (IR'95), pages 111-118, January 1995, and to the US patent 5,748,964 granted on 05/05/1998.

Compared with the second solution, the third solution presents the advantage of a much more rapid execution of the applets, since the virtual machine does not carry out any verification during execution.

The third solution, however, ~~features~~presents the drawback of a process of static verification of the code which is complex and expensive, both in terms of size of code necessary to conduct this process and in terms of size of random-access memory necessary to contain the intermediate results of the verification, and in terms of computation time. By way of illustrative example, the code verification ~~integrated into~~incorporated in the ~~Java~~JAVA JDK system marketed by SUN MICROSYSTEMS represents about 50 ~~bytes~~kbytes of machine code, and its consumption in terms of random-access memory is proportional to (Tp + Tr) x Nb, where Tp designates the maximum stack space, Tr designates the maximum number of registers and ~~N_p~~Nb designates the maximum number of ~~branching~~branch targets used by a ~~subprogram~~subroutine, also ~~widely designated by~~commonly called method, of the applet. These memory requirements greatly ~~exceeded~~exceed the capacities of the resources of ~~the majority of the~~most present-day ~~on-board~~embedded data-processing systems, especially of commercially available microprocessor cards.

Several variants of the third solution have been proposed, in which the ~~writer~~publisher of the applet sends to the verifier, in addition to the code of the applet, a certain amount of specific supplementary information such as precalculated data types or preestablished proof of correct data typing. For a more detailed description of the corresponding ~~operating modes~~procedures, reference could usefully be made to the articles published by Eva ROSE and Kristoffer ~~HOGSBRO~~HØGSBRO ROSE, "Lightweight Bytecode Verification", proceedings of the Workshop on Formal Underspinning of ~~Java~~JAVA, October 1998, and by George C. NECULA,

"Proof-Carrying Code", Proceedings of the 24th ACM Symposium on Principles of Programming Languages, pages 106-119, respectively.

This supplementary information makes it possible to verify the code more rapidly and slightly to reduce the size of the ~~code of the~~ verification program code but does not make it possible, however, to reduce the requirements for random-access memory, ~~or~~and even increases them, very substantially, in the case of the correct-data-typing preestablished-proof information.

SUMMARY OF THE INVENTION

The object of the present invention is to remedy the abovementioned drawbacks of the prior art.

In particular, one subject of the present invention is the implementation of a ~~protocol~~process for managing a downloaded program fragment, or applet, allowing execution of the latter by an ~~on-board~~embedded data-processing system having ~~few~~limited resources available, such as a microprocessor card.

Another subject of the present invention is also the implementation of a method of verifying a downloaded program fragment, or applet, in which a process of static verification of the code of the applet is conducted when it is downloaded, this process possibly being ~~aligned~~similar, at least in its principle, ~~with~~to the third solution described above, but ~~in~~into which new verification techniques are introduced, so as to allow execution of this verification within the ~~limits of values of~~ memory size and ~~of~~ computation speed value limits imposed by the microprocessor cards and other low-power ~~on-board~~embedded data-processing systems.

Another subject of the present invention is also the implementation of methods of transforming program fragments of conventional type obtained, for example, by the use of a ~~Java~~JAVA compiler ~~on~~, into standardized program fragments, or applets, satisfying, a priori, the

verification criteria of the verification method which is the subject of the invention, with a view to accelerating the process of verifying and executing ~~them at~~ the ~~level of~~latter in present-day microprocessor cards or ~~on board~~embedded data-processing systems.

Another subject of the present invention is, finally, the production of ~~on board~~embedded data-processing systems ~~allowing~~enabling the implementation of the abovementioned ~~protocol~~process for managing and ~~of the abovementioned~~ method of verifying a downloaded program fragment as well as of data-processing systems ~~allowing~~enabling the implementation of the methods of transforming conventional program fragments, or applets, into standardized program fragments, or applets, as mentioned above.

The ~~protocol~~process for managing a downloaded program fragment ~~on~~downloaded to a reprogrammable ~~on board~~embedded system, which is the subject of the present invention, applies especially to a microprocessor card ~~equipped~~provided with a rewritable memory. The program fragment consists of an object code, a series of instructions, executable by the microprocessor of the ~~on board~~embedded system by ~~way~~means of a virtual machine ~~equipped~~provided with an execution stack and with local variables or registers manipulated ~~via~~by these instructions and ~~making it possible~~used to interpret this object code. The ~~on board~~embedded system is interconnected ~~to~~with a terminal.

It is noteworthy in that it consists at least, at the level of the ~~on board~~embedded system, in detecting a command for downloading ~~of~~ the program fragment. On a positive response to the ~~stage~~step consisting in detecting a ~~downloading~~download command, it further consists in reading the object code constituting the program fragment and in temporarily storing this object code in the rewritable memory. The whole of the object code stored in memory is subjected to a verification process, instruction by instruction. The verification process consists at least in a

~~stage of~~step for initializing the type stack and the ~~table of~~ register ~~types~~type array representing the state of the virtual machine at the start of the execution of the temporarily stored object code and in a succession of ~~stages of verification~~steps for verifying, instruction by instruction, ~~of~~ the existence, for each current instruction, of a target, ~~branching~~branch-instruction target, target of an exception handler, and in a verification and an updating of the effect of the current instruction on the type stack and on the ~~table of~~ register ~~types.~~ type array. In the event of an unsuccessful verification of the object code, the ~~protocol~~process which is the subject~~,~~ of the invention consists in deleting the ~~momentarily recorded~~temporarily stored program fragment, ~~when omitting to record~~if the latter is not stored in the directory of available program fragments, and in sending an error code to the reader.

The method of verifying a program fragment downloaded ~~onto~~to an ~~on-board~~embedded system, which is the subject of the invention, applies ~~especially~~in particular to a microprocessor card equipped with a rewritable memory. The program fragment consists of an object code and includes at least one ~~subprogram~~subroutine, a series of instructions, executable by the microprocessor of the ~~on-board~~embedded system by ~~way~~means of a virtual machine ~~equipped~~provided with an execution stack and with operand registers manipulated by these instructions, and ~~making it possible~~used to interpret this object code. The ~~on-board~~embedded system is interconnected ~~to~~with a reader.

It is noteworthy in that, following the detection of a ~~downloading~~download command and the storage of the object code constituting the program fragment in the rewritable memory, it consists, for each ~~subprogram~~subroutine, in carrying out a ~~stage of~~step for initializing the type stack and the ~~table of~~ register ~~types~~type array by data representing the state of the virtual machine at the start of the execution of the temporarily stored object code, in carrying out a verification of

the temporarily stored object code instruction by instruction, by discerning the existence, for each current instruction, of~~.~~ a ~~branching~~branch-instruction target, of a target of an exception-handler call or of a target of a subroutine call, and in carrying out a verification and an updating of the effect of the current instruction on the data types of the type stack and of the ~~table of~~ register ~~type~~type array, on the basis of the existence of a ~~branching~~branch-instruction target, of a target of a subroutine call or of a target of an exception-handler call. The verification is successful when the ~~table of~~ register ~~type~~type array is not modified in the course of a verification of all the instructions, the verification process being carried out instruction by instruction until the ~~table of~~ register ~~type~~type array is stable, with no modification present. Otherwise the verification process is interrupted.

The method of transforming an object code of a program fragment into a standardized object code for this same program fragment, which is the subject of the present invention, applies to an object code of a program fragment in which the operands of each instruction belong to the data types manipulated by this instruction, the execution stack does not exhibit any overflow phenomenon and, for each ~~branching~~branch instruction, the type of the variables of the stack at this ~~branching~~branch is the same as at the targets of this ~~branching.~~ branch. The standardized object code obtained is such that the operands of each instruction belong to the data types manipulated by this instruction, the execution stack does not exhibit any overflow phenomenon and the execution stack is empty at each ~~branching~~branch-target instruction.

It is noteworthy in that it consists, for all the instructions of the object code, in annotating each current instruction with the data type of the execution stack before and after the execution of this instruction, the annotation data being calculated by means of an analysis of the data stream relating to this instruction, in detecting, within the instructions and within each current

instruction, the existence of ~~branchings~~branches for which the execution stack is not empty, the detection operation being carried out on the basis of the annotation data of the type of stack variables allocated to each current instruction. ~~In the presence of a~~ On detection of a non-empty execution stack, it further consists in inserting instructions to transfer stack variables on either side of these ~~branchings~~branches or of these ~~branching~~branch targets in order to empty the contents of the execution stack into temporary registers before this ~~branching~~branch and to reestablish the execution stack from the temporary registers after this ~~branching~~branch, and in not inserting any transfer instruction otherwise.

This method thus makes it possible to obtain a standardized object code for this same program fragment, in which the execution stack is empty at each ~~branching~~branch instruction and ~~branching~~branch-target instruction, in the absence of any modification ~~to~~of the execution of the program fragment.

The method of transforming an object code of a program fragment into a standardized object code for this same program fragment, which is the subject of the present invention, applies, moreover, to an object code of a program fragment in which the operands of each instruction belong to the data types manipulated by this instruction, and a operand of given type written into a register by an instruction of this object code is ~~reread~~read back from this same register by another instruction of this object code with the same given data type. The standardized object code obtained is such that the operands belong to the data types manipulated by this instruction, one and the same data type being allocated to the same register throughout the standardized object code.

It is noteworthy in that it consists, for all the instructions of the object code, in annotating each current instruction with the data type of the registers before and after the execution of this

instruction, the annotation data being calculated by means of an analysis of the data stream relating to this instruction, and in carrying out a reallocation of the original registers employed with different types, by dividing these original registers into separate standardized registers. One standardized register is allocated to each data type used. ~~Reupdating~~ A reupdating of the instructions which manipulate the operands which use the standardized registers is carried out.

The ~~protocol~~process for managing a program fragment, the method of verifying a program fragment, the methods of transforming object code of program fragments into standardized object code and the corresponding systems, which are the subjects of the present invention, find ~~an~~ application in the development of reprogrammable ~~onboard~~embedded systems, such as microprocessor cards, especially in the Java environment.

BRIEF DESCRIPTION OF THE DRAWINGS

They will be better understood on reading the description and on perusing the drawings below~~, in which, other than figures 1a and 1b relating to the prior art~~:

Figure 1a represents the architecture of a prior art embedded system.

Figure 1b represents the architecture of a prior art reprogrammable embedded system.

- Figure 2 represents a flow chart illustrating the ~~protocol~~process for managing a program fragment downloaded ~~onto~~to a reprogrammable ~~on-board~~embedded system,

Figure 3a represents, by way of illustration, a flow chart of a method of verifying~~,~~ a downloaded program fragment in accordance with the subject of the present invention,

Figure 3b represents a diagram illustrating data types and sub-typing relationships implemented by the method of managing and the method of verifying a downloaded program fragment, which is the subject of the present invention,

Figure 3c represents a detail of the verification method ~~as claimed in figure~~according to Figure 3a, relating to the managing of a ~~branching~~branch instruction,

Figure 3d represents a detail of the verification method ~~as claimed in figure~~according to Figure 3a, relating to the managing of a subroutine-call instruction,

Figure 3e represents a detail of the verification method ~~as claimed in figure~~according to Figure 3a, relating to the managing of an exception-handler target,

Figure 3f represents a detail of the verification method ~~as claimed in figure~~according to Figure 3a, relating to the managing of a target of incompatible ~~branchings~~branches,

Figure 3g represents a detail of the verification method ~~as claimed in figure~~according to Figure 3a, relating to the managing of an absence of ~~branching~~branch target,

Figure 3h represents a detail of the verification method ~~as claimed in figure~~according to Figure 3a, relating to the managing of the effect of the current instruction on the type stack,

Figure 3i represents a detail of the verification method ~~as claimed in figure~~according to Figure 3a, relating to the managing of ~~an~~a register read instruction ~~for reading a register~~,

Figure 3j represents a detail of the verification method ~~as claimed in figure~~according to Figure 3a, relating to the managing of ~~an~~a register write instruction ~~for writing to a register~~,

Figure 4a represents a flow chart illustrating a method of transforming an object code of a program fragment into a standardized object code for this same program fragment with a ~~branching~~branch instruction, respectively a ~~branching~~branch-target instruction, with an empty stack,

Figure 4b represents a flow chart illustrating a method of transforming an object code of a program fragment into a standardized object code for this same program fragment, making use of typed registers, a single specific data type being ~~attributed~~assigned to each register,

Figure 5a represents a detail of implementation of the transformation method illustrated in ~~figure~~Figure 4a,

Figure 5b represents a detail of implementation of the transformation method illustrated in ~~figure~~Figure 4b,

Figure 6 represents a functional diagram of the complete architecture of a system for ~~development of~~developing a standardized program fragment, and of a reprogrammable microprocessor card ~~allowing implementation of the protocol~~used to implement the process for managing and the method of verifying a program fragment in accordance with the subject of the present invention.

DESCRIPTION OF THE PREFERRED EMBODIMENTS

In general, it is indicated that the ~~protocol~~process for managing and the method of verifying and transforming a downloaded program fragment, which is the subject of the present invention, and the corresponding systems, are implemented ~~thanks to~~using a software architecture for the secure downloading and execution of applets on an ~~on-board~~embedded data-processing system with ~~few~~limited resources, such as, in particular, microprocessor cards.

In general, it is indicated that the description below concerns the application of the invention in the context of reprogrammable microprocessor cards of ~~JavaCard~~JAVACARD type, cf. documentation available electronically from the company SUN MICROSYSTEMS INC., ~~JavaCard~~JAVACARD Technology heading mentioned previously in the description.

However, the present invention is applicable to any ~~on-board~~embedded system which is reprogrammable by downloading an applet which is written in the code of a virtual machine including an execution stack, local variables or registers, and of which the execution model is strongly typed, each instruction of the code of the applet ~~applying~~being applied only to specific

data types. The ~~protocol~~process for managing a program fragment downloaded ~~onto~~to a reprogrammable ~~on-board~~embedded system, which is the subject of the present invention, will now be described in more detail with reference to ~~fig~~Fig. 2.

With reference to the abovementioned figure, it is indicated that the object code which makes up the program fragment or applet consists of a series of instructions which can be executed by the microprocessor of the ~~on-board~~embedded system by means of the abovementioned virtual machine. The virtual machine ~~makes it possible~~is used to interpret the abovementioned object code. The ~~on-board~~embedded system is interconnected ~~to~~with a terminal via, for instance, a serial link.

With reference to the abovementioned ~~fig~~Fig. 2, the management ~~protocol~~process which is the subject of the present invention consists at least, in the ~~on-board~~embedded system, in detecting a command to download this program fragment in a ~~stage 100~~step 100a, ~~100~~100b. Thus, ~~stage 100~~step 100a may consist of a ~~stage of~~step for reading the abovementioned command, and ~~stage 100~~step 100b may consist of a ~~stage of~~step for testing the command which has been read and verifying the existence of a ~~downloading~~download command.

On a positive response to the abovementioned ~~stage 100~~step 100a, ~~100~~100b ~~of~~for detecting a ~~downloading~~download command, the ~~protocol~~process which is the subject of the present invention ~~subsequently~~then consists in reading, at ~~stage~~step 101, the object code which makes up the relevant program fragment, and temporarily storing the abovementioned object code in the memory of the ~~on-board~~embedded data-processing system. The abovementioned temporary ~~storing~~storage operation can be executed either in the rewritable memory or, if appropriate, in the random-access memory of the ~~on-board~~embedded system, when ~~this~~the latter has sufficient

capacity. The ~~stage of~~step for reading the object code and temporarily storing it in the rewritable memory is designated ~~as loading the code of the~~load applet code in ~~fig~~Fig. 2.

The abovementioned ~~stage~~step is then followed by a ~~stage~~step 102 consisting in submitting the whole of the temporarily stored object code to a process of verification, instruction by instruction, of the abovementioned object code.

The verification process consists, at least in a ~~stage of~~step for initializing the type stack ~~of types~~ and the ~~table of~~ data ~~types~~type array representing the state of the virtual machine at the start of execution of the temporarily stored object code, and in a succession of ~~stages of~~steps for verifying, instruction by instruction, by discerning the existence, for each current instruction, designated I_i, of a target such as a ~~branching~~branch-instruction target designated ~~CIS, a target of~~CIB, an exception-handler call target or a target of a subroutine call. A verification and ~~update~~updating of the effect of the current instruction I_i on the type stack ~~of types~~ and on the ~~table of~~ register ~~types~~type array is carried out.

When the verification ~~has been~~103 is successful at ~~stage~~step 103a, the ~~protocol~~process which is the subject of the present invention consists in ~~recording~~storing, at ~~stage~~step 104, the downloaded program fragment in a directory of available program fragments, and in sending to the reader, at ~~stage~~step 105, a positive reception acknowledgment.

On the other hand, in the case of unsuccessful verification of the object code at ~~stage~~step 103b, the ~~protocol~~process which is the subject of the present invention consists in inhibiting, in a ~~stage~~step 103c, any execution on the ~~on board~~embedded system of the ~~momentarily recorded~~temporarily stored program fragment. The inhibition ~~stage~~step 103c can be implemented in various ways. As a nonlimiting example, this ~~stage~~step can consist in deleting, at ~~stage~~step 106, the ~~momentarily recorded~~temporarily stored program fragment, without

~~recording~~storing this program fragment in the directory of available program fragments and, at ~~stage~~step 107, in sending an error code to the reader.~~—Stages~~ Steps 107 and 105 can be implemented either sequentially after ~~stages~~steps 106 and 104 respectively, or in multitasking operation with them.

With reference to the same ~~fig~~Fig. 2, on a negative response to the ~~stage~~step consisting in detecting a ~~downloading~~download command at ~~stage 100~~step 100b, the ~~protocol~~process which is the subject of the present invention consists in detecting, in a ~~stage~~step 108, a command to select an available program fragment from a directory of program fragments and, on a positive response to ~~stage~~step 108, having detected the selection of an available program fragment, in calling, at ~~stage~~step 109, this selected available program fragment in order to execute it.~~—Stage~~ Step 109 is then followed by a ~~stage~~step 110 ~~of~~for executing the called available program fragment by ~~way~~means of the virtual machine, with no dynamic verification of variable types, rights of access to the objects which are manipulated by the called available program fragment, or overflow of the execution stack when each instruction is executed.

In the case where a negative response is obtained at ~~stage~~step 108, this ~~stage~~step consisting in detecting a command to select a called available program fragment, the ~~protocol~~process which is the subject of the present invention consists in proceeding, ~~in~~at a ~~stage~~step 111, to process the standard commands of the ~~on-board~~embedded system.

Regarding the absence of dynamic verification of type or rights of access to objects of, for instance, ~~JavaCard~~JAVACARD type, it is indicated that this absence of verification does not compromise the security of the card, because the code of the applet has necessarily successfully ~~passed~~undergone verification.

More specifically, it is indicated that the code verification which is carried out, ~~as claimed~~ in accordance with the method which is the subject of the present invention, on the microprocessor card or ~~on board~~embedded data-processing system is more selective than the customary verification of codes for the virtual ~~Java~~JAVA machine as described in the work entitled "The Java Virtual Machine Specification" ~~which was~~ mentioned previously in the description.

However, any code of the ~~Java~~JAVA virtual machine which is correct as far as the ~~traditional Java~~conventional JAVA verifier is concerned can be transformed into an equivalent code which is capable of ~~passing~~ successfully undergoing the code verification which is carried out on the microprocessor card.

Whereas it is possible to imagine writing directly ~~Java~~JAVA codes which satisfy the ~~abovementioned~~ verification criteria mentioned previously in the context of implementing the ~~protocol~~process which is the subject of the present invention, a noteworthy object of the latter is also the implementation of a method of automatic transformation of any standard ~~Java~~JAVA code into a standardized code for the same program fragment, necessarily satisfying the verification criteria implemented as mentioned above. The method of transformation into standardized code, and the corresponding system, will be described in detail ~~subsequently~~later in the description.

A more detailed description of the method of verifying a program fragment, or applet, in accordance with the subject of the present invention, will now be given with reference to ~~fig~~Fig. 3a and the subsequent figures.

In general, it is indicated that the verification method which is the subject of the present invention can be implemented either as part of the ~~protocol~~process for managing a program

fragment which is the subject of the present invention as described above with reference to ~~fig~~Fig. 2, or independently, to provide whatever verification process is judged necessary.

In general, it is indicated that a program fragment is made up of an object code including at least one ~~subprogram~~subroutine, more commonly ~~designated~~called a method, and is made up of a series of instructions which can be executed by the microprocessor of the ~~on-board~~embedded system ~~via~~by means of the virtual machine.

As shown in ~~fig~~Fig. 3a, the verification method consists, for each ~~subprogram~~subroutine, in carrying out a ~~stage~~step 200 ~~of~~for initializing the type stack ~~of types~~ and the ~~table of~~ register ~~types~~type array of the virtual machine by data representing the state of this virtual machine at the start of execution of the object code which is the subject of the verification. This object code can be stored temporarily as described above with reference to implementation of the ~~protocol~~process which is the subject of the present invention.

The abovementioned ~~stage~~step 200 is then followed by a ~~stage~~step 200a consisting in positioning the reading of the current instruction ~~It~~I_i, index i, on the first instruction of the object code. ~~Stage~~ Step 200a is followed by a ~~stage~~step 201 consisting in carrying out a verification of the abovementioned object code, instruction by instruction, by discerning the existence, for each current instruction, designated I_i, of a ~~branching~~branch-instruction target ~~GIB~~CIB, of a target of an exception-handler call, designated CEM, or of a target of a subroutine call CSR.

The verification ~~stage~~step 201 is followed by a ~~stage~~step 202 ~~of~~for verifying and updating the effect of the current instruction I_i on the data types of the type stack ~~of types~~ and of the ~~table of~~ register ~~types~~type array, as a function of the existence, for the current instruction which is pointed ~~at~~to by another instruction, of a ~~branching~~branch-instruction target ~~GIB~~CIB, of a target of a subroutine call CSR or of a target of an exception-handler call CEM.

~~Stage~~Step 202 for the current instruction I_i is followed by a ~~stage~~step 203 to test whether the last instruction has been reached, the test written as:

I_i = last instruction of the object code?

On a negative response to test 203, the process passes to the next instruction 204, written i = i+1, and ~~to the~~on return to ~~stage~~step 201.

It is indicated that the abovementioned verification, at ~~stage~~step 202, ~~has been~~is successful when the ~~table of~~ register ~~types~~type array is not modified during verification of all the instructions I_i which make up the object code. For this purpose, a test 205 of the existence of a stable state of the ~~table of~~ register ~~types and~~type array is provided. This test is written:

∃? Stable state of ~~table of~~ register ~~types~~type array.

On a positive response to test 205, the verification has been successful.

On the other hand, in the case where no absence of modification is noticed, the verification process is repeated and reinitiated by returning to ~~stage~~step 200a. It is demonstrated that the process is guaranteed to end after a maximum of NrxH iterations, where Nr designates the number of registers used and H designates a constant depending on the subtyping relation.

Various indications concerning the types of variables ~~which are~~ manipulated in the course of the verification process described above with reference to ~~fig~~Fig. 3a will now be given with reference to ~~fig~~Fig. 3b.

The abovementioned variable types include at least class identifiers corresponding to object classes ~~which are~~ defined in the program fragment which is subjected to verification, numeric variable types including at least a ~~type~~ short type, an integer coded on p bits, where the value of p can be 16, and a type for the return address of a jump instruction JSR, this address type being ~~identified as~~ denoted retaddr, a ~~type~~ null type relating to references of null objects, ~~a type~~an

object type relating to the objects proper, a specific type ⊥ representing the intersection of all the types and corresponding to the ~~value~~ zero ~~null~~value, another specific type T representing the union of all the types and corresponding to any type of values.

With reference to ~~fig~~Fig. 3b, it is indicated that all the abovementioned variable types verify a subtyping relation:

object ~~∈~~ ∈ T;

short, retaddr ~~∈~~ ∈ T;

⊥ ~~∈~~ ∈ null, short, retaddr

A more specific example of a process of verification as illustrated in fig. 3a will now be given, with reference to a ~~first example of a~~ data structure example, which is shown in ~~table~~array T1 in the annex.

The abovementioned example concerns an applet written in ~~Java~~JAVA code.

The verification process accesses the code of the ~~subprograms~~subroutine which forms the applet which is subjected to verification via a pointer to instruction I_i which is being verified.

The verification process records the size and type of the execution stack at the current instruction I_i corresponding to saload in the example of the ~~above-mentioned Table~~abovementioned Array T1.

The verification process then ~~records~~stores the size and type of the execution stack at the current instruction in the type stack ~~of types~~ via its type stack pointer.

As mentioned above in the description, this type stack ~~of types~~ reflects the state of the execution stack of the virtual machine at the current instruction I_i. In the example shown in ~~table~~array T1, at the time of the future execution of instruction I_i, the stack will contain three entries: a reference to an object of class C, a reference to ~~a table of integers~~an integer array

coded on p = 16 bits, the type short [ ], and an integer of p = 16 bits of type short. This is also shown in the type stack, which also contains three entries: C, the type of the objects of class C, short [], the type of ~~tables~~the arrays of integers p = 16 bits and short, the type of integers p = 16 bits.

Another noteworthy data structure consists of ~~a table of~~an register ~~types~~type array, this ~~table~~array reflecting the state of the registers, that is to say of the registers which store the local variables, of the virtual machine.

Continuing the example indicated in ~~table~~array T1, it is indicated that entry 0 of the ~~table of~~ register ~~types~~type array contains type C, i.e. at the time of the future execution of the current instruction I_i = saload, register 0 is guaranteed to contain a reference to an object of class C.

The various types which are manipulated during the verification and stored in the ~~table of~~ register ~~types~~type array and in the type stack are represented in ~~fig~~Fig. 3b. These types include:

- class identifiers CB corresponding to specific object classes which are defined in the applet;
- ~~base~~basic types, such as short~~, an~~ integer coded on p = 16 bits, ~~int~~int1 and int2, the most and least significant p bits respectively of integers coded on, e.g., 2p bits, or retaddr, the return address of an instruction as mentioned above;
- the type null, representing the references of null objects.

Regarding the subtyping relation, it is indicated that a type T1 is a subtype of a type T2 if ~~every~~any valid value of type T1 is also a valid value of type T2. The subtyping between class identifier reflects the inheritance hierarchy between classes of the applet. On the other types, subtyping is defined by the lattice shown in fig. 3b, where $\perp$ is a subtype of all the types and all the types are subtypes of T.

The sequence of the process of verifying a ~~subprogram~~subroutine which forms an applet is as follows, referring to the abovementioned ~~table~~array T1.

The verification process is carried out independently on each ~~subprogram~~subroutine of the applet. For each ~~subprogram~~subroutine, the process carries out one or more verification passes on the instructions of the relevant ~~subprogram.~~ subroutine. The pseudocode of the verification process is given in ~~table~~array T2 in the annex.

The process of verifying a ~~subprogram~~subroutine begins with initializing the type stack and the ~~table of~~ register ~~types~~type array shown in ~~table~~array T1, this initialization reflecting the state of the virtual machine at the start of execution of the ~~subprogram~~subroutine being examined.

The type stack is initially empty, the stack pointer equals zero, and the register types are initialized with the types of the parameters of the ~~subprogram~~subroutine, illustrating the fact that the virtual machine passes the parameters of this ~~subprogram~~subroutine in these registers. The register types ~~which are~~ allocated by the ~~subprogram~~subroutine are initialized to data types $\perp$, illustrating the fact that the virtual machine initializes these registers to zero at the start of execution of the ~~subprogram~~subroutine.

Next, one or more verification passes on the instructions and on each current instruction I_i of the ~~subprogram~~subroutine are carried out.

At the end of the implemented verification pass, or of a succession of passes for ~~instance~~example, the verification process determines whether the register types contained in the ~~table of~~ register ~~types shown in table~~type array represented in array T1 of the annex have changed during the verification pass. In the absence of change, verification is terminated and a success

code is returned to the main program, which makes it possible to send the positive reception acknowledgment at ~~stage~~step 105 of the management ~~protocol~~process shown in ~~fig~~Fig. 2.

If a change to the abovementioned ~~table of~~ register ~~type~~type array is present, the verification process repeats the verification pass until the register types contained in the ~~table of~~ register ~~types is~~type array are stable.

The sequence proper of a verification pass which is carried out one or more times until the ~~table of~~ register ~~type~~type array is stable will now be described with reference to ~~figs~~Figs. 3c to 3j.

For each current instruction I_i, the following verifications are carried out:

With reference to ~~fig~~Fig. 3a at ~~stage~~step 201, the verification process determines whether the current instruction I_i is the target of a ~~branching~~branch instruction, a subroutine call or an exception-handler call, as mentioned above. This verification is carried out by examining the ~~branching~~branch instructions in the code of the ~~subprogram~~subroutine and the exception handlers associated with this ~~subprogram~~subroutine.

With reference to ~~fig~~Fig. 3c which begins with ~~stage~~step 201, when the current instruction I_i is the target of a ~~branching~~branch instruction, this condition being implemented by a test 300 designated by I_i = CIB, this ~~branching~~branch being unconditional or conditional, the verification process checks that the type stack is empty at this point of the ~~subprogram~~subroutine by a test 301. On a positive response to the test 301, the verification process is continued by a context continuation ~~stage~~step marked continue A. On a negative response to the test 301, the type stack not being empty, the verification fails and the applet is rejected. This failure is represented by the Failure ~~stage~~step.

With reference to ~~fig~~Fig. 3d which begins with the continue A ~~stage~~step, when the current instruction I_i is the target of a subroutine call, this condition being implemented by a test 304 I_i = CSR, the verification process verifies, in a test 305, that the previous instruction I_{i-1} does not continue in sequence. This verification is implemented by a test ~~stage~~step 305 when the previous instruction is an unconditional ~~branching~~branch, a subroutine return or a ~~raising~~withdrawal of an exception. The test at ~~stage~~step 305 is marked as follows:

$I_{i-1} = IB_{unconditional}$, return RSR or ~~raising~~withdrawal L-EXCEPT.

On a negative response to test 305, the verification process fails in a Failure ~~stage~~step. On the other hand, on a positive response to test 305, the verification process reinitializes the type stack 306 in such a way that it contains exactly one entry of retaddr type, the return address of the abovementioned subroutine. If the current instruction I_i at ~~stage~~step 304 is not the target of a subroutine call, the verification process is continued in the context at the continue B ~~stage~~step.

With reference to ~~fig~~Fig. 3e, when the current instruction I_i is the target of an exception handler, this condition being implemented by a test 307 marked I_i = CEM, where CEM designates the target of an exception handler, this condition is implemented by a test 307, marked:

I_i = CEM.

On a positive response to test 307, the process verifies that the previous instruction is an unconditional ~~branching~~branch, a subroutine return or a ~~raising~~withdrawal of exceptions by a test 305, marked:

$I_{i-1} = IB_{unconditional}$, return RSR or ~~raising~~withdrawal L-EXCEPT.

On a positive response to test 305, the verification process proceeds to reupdate the type stack, at a ~~stage~~step 308, ~~by entering~~with an exception types entry, marked EXCEPT type,

~~stage~~step 308 being followed by a context continuation ~~stage~~step, continue C. On a negative response to test 305, the verification process fails ~~by~~with the ~~stage~~step marked Failure. The program fragment is then rejected.

With reference to ~~fig~~Fig. 3f, when the current instruction I_i is the target of multiple incompatible ~~branchings~~branches, this condition is implemented by a test 309, which is marked:

I_i = incompatible XIBs

the incompatible ~~branchings~~branches being, for instance, an unconditional ~~branching~~branch and a subroutine call, or even two different exception handlers. On a positive response to test 309, the ~~branchings~~branches being incompatible, the verification process fails ~~by~~with a ~~stage~~step marked Failure and the program fragment is rejected. On a negative response to test 309, the verification process is continued ~~by~~with a ~~stage~~step marked continue D. Test 309 begins with the continue C ~~stage which was~~step mentioned previously in the description.

With reference to ~~fig~~Fig. 3g, when the current instruction I_i is not the target of any ~~branching~~branch, this condition being implemented by a test 310 beginning with the abovementioned continue D, this test being marked

I_i ∃? ~~branching~~branch targets,

where ∃ ~~designates~~denotes the existence symbol,

the verification process continues on a negative response to the test 310 by ~~passing~~going on to an update of the type stack at ~~stage~~step 311, ~~stage~~step 311 and the² positive response to test 310 being followed by a context continuation ~~stage~~step at ~~stage~~step 202, which is described above in the description with reference to ~~fig~~Fig. 3a.

A more detailed description of the ~~stage of~~step for verifying the effect of the current instruction on the type stack at the abovementioned ~~stage~~step 202 will now be given with reference to ~~fig~~Fig. 3h.

According to the abovementioned figure, this ~~stage~~step can include at least one ~~stage~~step 400 ~~of verification~~for verifying that the type execution stack ~~includes~~contains at least as many entries as the current instruction includes operands. This test ~~stage~~step 400 is marked:

Nbep ≥= NOpi

where Nbep ~~designates~~denotes the number of ~~entries of the~~ type stack entries and NOpi ~~designates~~denotes the number of operands ~~included~~contained in the current instruction.

On a positive response to test 400, this test is followed by a ~~stage~~step 401a ~~of~~for unstacking the type stack, and ~~of~~for verifying 401b that the types of the entries at the top of the stack are subtypes of the types of the operands of the abovementioned current instruction. At test ~~stage~~step 401a, the operand types of the instruction i are marked TOpi, and the types of the entries at the top of the stack are marked Targs.

At ~~stage~~step 401b, the verification corresponds to a verification of the subtyping relation Targs subtype of TOpi.

On a negative response to tests 400 and 401b, the verification process fails, which is shown by access to the Failure ~~stage~~step. On the other hand, on a positive response to test 401b, the verification process is continued, and consists in carrying out:

- ~~–~~A ~~stage of~~step for verifying the existence of a sufficient memory space on the type stack to proceed to stack the results of the current instruction. This verification ~~stage~~step is implemented by a test 402, marked:

Stack-space ≥ Results-space

where each side of the inequality ~~designates~~denotes the corresponding memory space.

On a negative response to test 402, the verification process fails, which is shown by the Failure ~~stage~~step. On the other hand, on a positive response to test 402, the verification process then proceeds to stack the data types ~~which are~~ assigned to the results in a ~~stage~~step 403, the stacking being done on the ~~stack of~~ data types ~~which are~~stack assigned to these results.

As a nonlimiting example, it is indicated that to implement ~~fig~~Fig. 3h for verifying the effect of the current instruction on the type stack, for a current instruction consisting of a ~~Java~~JAVA saload instruction corresponding to reading an integer element coded on p = 16 bits in ~~a table of integers~~an integer array, this ~~table of integers~~integer array being defined by the ~~table of integers~~integer array and an integer index in this ~~table~~array, and the result by the integer which is read at this index in this ~~table~~array, the verification process checks that the type stack contains at least two elements, that the two elements at the top of the type stack are subtypes of short [ ] and short respectively, proceeds to the unstacking process and then to the process of stacking the data type short as the result type.

Additionally, with reference to ~~fig~~Fig. 3i, to implement the ~~stage of~~step for verifying the effect of the current instruction on the type stack, when the current instruction I_i is a read instruction, marked IR, of a register of address n, this condition being implemented by a test 404 marked $I_i = IR_n$, on a positive response to the abovementioned test 404, the verification process consists in verifying the data type of the result of this ~~reading~~read, in a ~~stage~~step 405, by reading the entry n in the ~~table of~~ register ~~type~~type array, then in determining the effect of the current instruction I_i on the type stack by an operation 406a of unstacking the entries of the stack corresponding to the operands of this current instruction and by stacking 406b the data type of this result. The operands of the instruction I_i are marked OP_i. ~~Stages~~ Steps 406a and 406b are

followed by a return to the context continuation, continue F. On a negative response to test 404, the verification process ~~i~~is continued by the context continuation, continue F.

With reference to ~~fig~~Fig. 3j, when the current instruction I_i is a write instruction, marked IW, of a register of address n, this condition being implemented by a test marked $I_i = IW_m$, on a positive response to test 407, the verification process consists in determining, in a ~~stage~~step 408, the effect of the current instruction on the type stack and the type t of the operand ~~which is~~ written in the register of address n, then, in a ~~stage~~step 409, in replacing the type entry of the ~~table of~~ register ~~types~~type array at address n ~~by~~with the type immediately above the previously stored type and above the type t of the operand ~~which is~~ written in the register of address n. ~~Stage~~Step 409 is followed by a return to the context continuation, continue 204. On a negative response to test 407, the verification process is continued by a context continuation, continue 204.

As an example, when the current instruction I_i corresponds to writing a value of type D into a register of address 1, and the type of register 1 before verification of the instruction was C, the type of register 1 is replaced by the type object, which is the smallest type ~~which is~~ higher than C and D in the lattice of types shown in ~~fig~~Fig. 3b.

In the same way, as an example, when the current instruction I_i is a read of an instruction aload-0 consisting in stacking the ~~contents~~content of register 0, and entry 0 of the ~~table of~~ register ~~types~~type array is C, the verifier stacks C onto the type stack.

An example of verifying a ~~subprogram~~subroutine written in a ~~Java~~JAVA environment will now be given, with reference to tables T3 and T4 in the annex.

~~Table~~Array T3 represents a specific ~~JavaCard~~JAVACARD code corresponding to the Java ~~subprogram which is~~subroutine included in this ~~table~~array.

~~table~~Array T4 shows the ~~contents~~content of the ~~table of~~ register ~~types~~type array and of the type stack before verification of each instruction. The type constraints on the operands of the various instructions are all observed. The stack is empty both after the instruction 5 to branch to instruction 9, symbolized by the arrow, and before the abovementioned ~~branching~~branch target 9. The type of register 1, which was initially ⊥, becomes null, the upper bound of null and ⊥, when instruction 1 to store a value of type null in register 1 is examined, then becomes of type short[ ], the upper bound of types short[ ] and null, when instruction 8 to store a value of type short [ ] in register 1 is processed. ~~The~~ Since the type of register 1 ~~having~~has changed during the first verification pass, a second pass is carried out, ~~distributing~~this time starting from the register types ~~which were~~ obtained at the end of the first. This second verification pass is successful, just like the first, and does not change the register types. The verification process thus terminates successfully.

Various examples of cases of failure of the verification process on four examples of incorrect code will now be given with reference to ~~table~~array T5 in the annex:

- At point a) of ~~table~~array T5, the purpose of the code given as an example is to attempt to ~~fabricate~~construct an invalid object reference using an arithmetic process on pointers. It is rejected by verification of the types of arguments of instruction 2 sadd, which requires these two arguments to be of type short.

- At points b) and c) of ~~table~~array T5, the purpose of the code is to carry out two attempts to ~~transform~~convert any integer into an object reference. At point b) , register 0 is used simultaneously with type short, instruction 0, and with type null, instruction 5. Consequently, the verification process assigns type T to ~~record~~register 0, and detects a type error when register 0 is returned as a result of type object at instruction 7.

- At point c) of ~~table~~array T5, a set of ~~branchings~~branches of type "if . . . then . . . else . . ." is used to leave at the top of the stack a result which consists of either an integer or an object reference. The verification process rejects this code because it detects that the stack is not empty at the ~~branching~~branch from instruction 5 to instruction 9, symbolized by the arrow.

- Finally, at point d) of ~~table~~array 5, the code contains a loop which, ~~at~~on each iteration, has the effect of stacking an additional integer on the top of the stack, and thus causing a stack overflow after a certain number of iterations. The verification process rejects this code, ~~noticing~~observing that the stack is not empty at the backward ~~branching~~branch from instruction 8 to instruction 0, symbolized by the return arrow, the stack not being empty at a ~~branching~~branch point.

The various examples given above with reference to tables T3, T4 and T5 show that the verification process, which is the subject of the present invention, is particularly ~~efficient~~effective, and that it applies to applets, and in particular to ~~subprograms~~subroutines thereof, for which the conditions of stack type, or respectively of the empty character of the type stack, previously~~, and to~~ on the ~~branching~~branch or branch target instructions~~ or branching targets~~, are satisfied.

Obviously, such a verification process implies writing object codes which satisfy these criteria, these object codes possibly corresponding to the ~~sub-programs~~subroutine in the abovementioned ~~table~~array T3.

However, and in order to ensure the verification of existing applets and ~~subprograms~~subroutines of applets which do not necessarily satisfy the verification criteria of the method which is the subject of the present invention, in particular regarding applets and ~~subprograms which are~~subroutines written in the Java environment, the purpose of the present

invention is to establish methods of transforming these applets or ~~subprograms~~subroutines into standardized applets or program fragments~~, making it possible to undergo~~ that can successfully undergo the verification tests of the verification method which is the subject of the present invention and of the management ~~protocol~~process which implements such a method.

For this purpose, the subject of the invention is the implementation of a method and a program for transforming a ~~traditional~~conventional object code forming an applet, it being possible to implement this method and this transformation program~~,~~ outside an ~~on-board~~embedded system or microprocessor card~~,~~ when the relevant applet is created.

The method of transforming code into standardized code, which is the subject of the present invention, will now be described in the framework of the ~~Java~~JAVA environment, as a purely illustrative example.

The JVM codes ~~which are~~ produced by existing ~~Java~~JAVA compilers satisfy various criteria, which are stated below:

~~Cl~~C1: the arguments of each instruction ~~actually~~do belong to the types which this instruction expects;

C2: the stack does not overflow;

C'3: for each ~~branching~~branch instruction, the type of the stack at this ~~branching~~branch is the same as at the possible targets for this ~~branching;~~branch;

C'4: a value of type t ~~which is~~ written into a register at one point of the code and ~~reread~~read back from the same register at another point of the code is always ~~reread~~read back with the same type t;

The implementation of the verification method which is the subject of the present invention ~~implies that~~entails criteria C'3 and C'4, ~~which are~~ verified by the object code ~~which is~~ submitted for verification, ~~be~~being replaced by criteria C3 and C4 below:

C3: the stack is empty at each ~~branching~~branch instruction and at each ~~branching~~branch target;

C4: the same register is used with one and the same type throughout the code of a ~~subprogram.~~ subroutine.

With reference to the abovementioned criteria, it is indicated that Java compilers guarantee only the weaker criteria C'3 and C'4. The verification process which is the subject of the present invention and the corresponding management ~~protocol~~process in fact guarantee more restrictive criteria C3 and C4, making it possible to ensure the security of execution and management of applets.

The concept of standardization, covering the transformation of codes into standardized codes, can present various aspects, ~~to~~inasmuch as the ~~extent that~~ replacement of criteria C'3 and C'4 by criteria C3 and C4, in ~~conformity~~accordance with the verification process which is the subject of the present invention, can be implemented ~~independently~~separately, to ensure that the stack is empty at each ~~branching~~branch instruction and at each ~~branching~~branch target, and respectively that the registers which the applet opens are typed, and a single data type which is assigned for execution of the relevant applet corresponds to each open register, or, on the other hand, jointly, to satisfy the whole of the verification process which is the subject of the present invention.

The method of transforming an object code into standardized object code ~~as claimed in~~according to the invention will consequently be described ~~as claimed in~~according to two

distinct ~~implementation modes~~embodiments, a first ~~implementation mode~~embodiment corresponding to the transformation of an object code which satisfies criteria C1, C2, C'3, C'4 into a standardized object code which satisfies criteria ~~C1,~~Cl, C2, C3, C'4 corresponding to a standardized code with an empty ~~branching~~branch instruction or ~~branching~~branch target, then, ~~as claimed in~~according to a second ~~implementation mode~~embodiment, in which the ~~traditional~~conventional object code which satisfies the same initial criteria is transformed into a standardized object code which satisfies criteria C1, C2, C'3, C4, for instance corresponding to a standardized code using typed registers.

The first ~~implementation mode~~embodiment of the code transformation method which is the subject of the present invention will now be described with reference to ~~fig~~Fig. 4a. In the ~~implementation mode which is~~embodiment shown in ~~fig~~Fig. 4a, the initial ~~traditional~~conventional code is considered to satisfy criteria C1+C2+C'3, and the standardized code ~~which is~~ obtained as the result of the transformation is considered to satisfy criteria C1+C2+C3.

According to the abovementioned figure, the transformation method consists, for each current instruction I_i of the code or of the ~~subprogram~~subroutine, in annotating each instruction, in a ~~stage~~step 500, with the data type of the stack before and after execution of this instruction. The annotation data is marked AI_i and is associated by the relation $I_i \leftrightarrow AI_i$ ~~with~~ in the relevant current instruction. The annotation data is calculated by means of an analysis of the data stream relating to this instruction. The data types before and after execution of the instruction are marked tbe_i and tae_i respectively. Calculation of annotation data by analysis of the data stream is a ~~traditional~~conventional calculation ~~which is~~ known to those skilled in the art, and will therefore not be described in detail.

The operation ~~which is implemented~~carried out at ~~stage~~step 500 is illustrated in ~~table~~array T6 in the annex, in which, for an applet or ~~subprogram of an~~ applet subroutine including 12 instructions, the annotation data AI_i made up of the types of registers and the types of the stack ~~are~~is introduced.

The abovementioned ~~stage~~step 500 is then followed by a ~~stage~~step 500a consisting in positioning the index i on the first instruction I_i = ~~I_i. Stage~~I_1. Step 500a is followed by a ~~stage~~step 501 consisting in detecting, among the instructions and in each current instruction I_i, the existence of ~~branchings~~branches marked IB or of ~~branching~~branch targets CIB for which the execution stack is not empty. This detection 501 is implemented by a test which is carried out on the basis of the annotation data AI_i of the type of stack variables allocated to each current instruction, the test being marked for the current instruction:

I_i is an IB or CIB and stack (AI) ~~?~~$\neq$ empty.

On a positive response to test 501, i.e. in the presence of detection of a non-empty execution stack, the abovementioned test is followed by a ~~stage~~step consisting in inserting instructions to transfer stack variables on either side of these ~~branchings~~branches IB or ~~branching~~branch targets CIB, in order to empty the content of the execution stack into temporary registers before this ~~branching~~branch and to reestablish the execution stack from the temporary registers after this ~~branching.~~ branch. The insertion ~~stage~~step is marked 502 in ~~fig~~Fig. 4a. It is followed by a ~~stage~~step 503 to test the reaching of the last instruction, marked

I_i = last instruction?

On a negative response to ~~the~~ test 503, an increment 504 i=i+1 is carried out, to go on to the next instruction and return to ~~stage~~step 501. On a positive response to test 503, an End ~~stage~~step is initiated. On a negative response to test 501, the transformation method is continued by a

~~branching~~branch to ~~stage~~step 503 in the absence of insertion of a transfer instruction.—~~Implementation~~ The implementation of the method of transforming a ~~traditional~~conventional code into a standardized code with ~~branching~~branch instruction with empty stack as represented in ~~fig~~Fig. 4a makes it possible to obtain a standardized object code for the same initial program fragment in which the stack of stack variables is empty at each ~~branching~~branch instruction and each ~~branching~~ branch target instruction, in the absence of any modification to the execution of the program fragment. In the case of a Java environment, the instructions to transfer data between stack and register are the load and store instructions of the Java virtual machine.

Returning to the example introduced in ~~table~~array T6, the transformation method detects a ~~branching~~branch target where the stack is not empty at instruction 9.~~The method is then to insert an~~ An instruction istore 1 is then inserted before the ~~branching~~branch instruction 5 which leads to the ~~above-mentioned~~abovementioned instruction 9, in order to save the content of the stack in register 1 and ensure that the stack is empty at the time of the ~~branching.~~ branch. Symmetrically, an instruction iload 1 is inserted before the instruction target 9, to reestablish the content of the stack exactly as it was before the ~~branching.~~ branch. Finally, an instruction istore 1 is inserted after instruction 8 to ensure that the stack is balanced on the two paths which lead to instruction 9. The result of the transformation carried out in this way into a standardized code is shown in ~~table~~array T7.

The second ~~implementation mode~~embodiment of the transformation method which is the subject of the present invention will now be described with reference to ~~fig~~Fig. 4b in the case in which the initial ~~traditional~~conventional object code satisfies criteria ~~C1~~Cl+C'4 and the standardized object code satisfies criteria C1+C4.

With reference to the abovementioned ~~fig~~Fig. 4b, it is indicated that the method, in this ~~implementation mode~~embodiment, consists in annotating, ~~as claimed in a stage~~according to a step 500 which is approximately the same as that ~~which is~~ shown in ~~fig~~Fig. 4a, each current instruction I_i with the data type of ~~register data~~the registers before and after execution of this instruction. In the same way, the annotation data ~~All~~AI_i is calculated by means of an analysis of the data stream relating to this instruction.

The annotation ~~stage~~step 500 is then followed by a ~~stage~~step consisting in carrying out a reallocation of the registers, the ~~stage~~step marked 601, by detecting the original registers employed with different types, ~~by~~and dividing these original registers into separate standardized registers, one standardized register being allocated to each data type used. ~~Stage~~ Step 601 is followed by a ~~stage~~step 602 ~~of~~for reupdating the instructions which manipulate the operands which use the abovementioned standardized registers. ~~Stage~~Step 602 is followed by a context continuation ~~stage~~step 302.

With reference to the example given in ~~table~~array T6, it is indicated that the transformation method detects that the register of rank 0, marked ~~r0~~,r0, is used with the two types, object, instructions 0 and 1, and int, instruction 9 and following. The ~~method is then to divide the~~ original register ~~r0~~r0 is then divided into two registers, register 0 for the use of object types and register 1 for uses of int type. References to ~~record~~register 0 of int type are then rewritten by transforming them into references to ~~record~~register 1, the standardized code obtained being shown in ~~table~~array T8 in the annex.

It is noted, in a nonlimiting way, that in the example ~~which is~~ introduced with reference to the abovementioned ~~table~~array T8, the new register 1 is used simultaneously for

standardization of the stack and for the creation of typed registers by dividing of register 0 into two registers.

The method of transforming a ~~traditional~~conventional code into a standardized code with ~~branching~~branch instruction with empty stack as described in ~~fig~~Fig. 4a will now be described in more detail in a preferred, nonlimiting ~~implementation mode~~embodiment, in relation to ~~fig~~Fig. 5a.

This ~~implementation mode~~embodiment concerns ~~stage~~step 501, consisting in detecting, within the instructions and within each current instruction I_i, the existence of ~~branching~~branch IB, or respectively of ~~branching~~branch target CIB, for which the stack is not empty.

Following the determination of target instructions where the stack is not empty, this condition being marked at ~~stage~~step 504a, I_i stack ~~? ≠~~empty, the transformation process consists in associating with these instructions, at the abovementioned ~~stage~~step 504a, a set of new registers, one ~~per~~for each stack location which is active at these instructions. Thus, if i ~~designates~~denotes the rank of a ~~branching~~branch target of which the associated stack type is not empty and is of type ~~tpl_i~~$tp1_i$ to tpn_i with $n > 0$, stack not empty, the transformation process allocates n new, as yet unused, registers, ~~r_i~~r_1 to r_n, and associates them with the corresponding instruction i. This operation is implemented at ~~stage~~step 504a.

~~Stage~~Step 504a is followed by a ~~stage~~step 504 consisting in examining each detected instruction of rank i and identifying, in a test ~~stage~~step 504, the existence of a ~~branching~~branch target CIB or of a ~~branching~~branch IB. ~~Stage~~ Step 504 is shown in the form of a test designated by:

~~3~~∃? CIB, IB and I_i = CIB.

In the case ~~that~~where the instruction of rank i is a ~~branching~~branch target CIB represented by the preceding equality, and ~~that~~where the stack of stack variables at this instruction is not empty, i.e. with a positive response to test 504, for ~~every~~any preceding instruction of rank i-1 consisting of a ~~branching~~branch, a ~~raising~~withdrawal of an exception or a program return, this condition is implemented at test ~~stage~~step 505, designated by:

I_{i-1} = IB, EXCEPT ~~raising~~withdrawal, Prog, return.

The detected instruction of rank i is only accessible by a ~~branching.~~ branch. On a positive response to the abovementioned test 505, the transformation process consists in carrying out a ~~stage~~step 506 consisting in inserting a set of load instructions of load type from the set of new registers before the relevant detected instruction of rank i. The insertion operation 506 is followed by a redirection 507 of all ~~branchings~~branches to the detected instruction of rank i, to the first inserted load instruction load. The insertion and redirection operations are shown in ~~table~~array T9 in the annex.

For ~~every~~any preceding instruction of rank i-1 continuing in sequence, i.e. when the current instruction of rank i is accessible simultaneously by a ~~branching~~branch and from the preceding instruction, this condition being implemented by test 508 and symbolized by the relations:

$$I_{i-1} \rightarrow I_i$$

~~And~~
and

$$IB \rightarrow I_i$$

the transformation process consists, in a ~~stage 509 of~~step 509, in inserting a set of ~~backup~~store instructions ~~store~~ to ~~back up~~save to the set of new registers before the detected instruction of rank

i, and a set of load instructions ~~load~~ to load from this set of new registers. ~~Stage~~ Step 509 is then followed by a ~~stage~~step 510 ~~of redirection of~~for redirecting all the ~~branchings~~branches to the detected instruction of rank i to the first inserted load instruction ~~load~~.

In the case ~~that~~where the detected instruction of rank i is a ~~branching~~branch to a determined instruction, for any detected instruction of rank i consisting of an unconditional ~~branching~~branch, this condition being implemented by a test 511 marked:

$I_{i+1} = IB_{uncondit.}$

the transformation process as shown in ~~fig~~Fig. 5a consists in inserting at a ~~stage~~step 512, on a positive response to test 511, before the detected instruction of rank i, multiple ~~backup~~store instructions ~~store~~. The transformation process inserts before the instruction i the n store instructions ~~store~~ as shown in ~~table~~array T11 as an example. The store instructions ~~store~~ address registers ~~r~~r_1 to r_n, where n ~~designates~~denotes the number of registers. Thus the ~~backup~~store instruction is associated with each new register.

For every detected instruction of rank i consisting of a conditional ~~branching~~branch, and for a number mOp, greater than 0, of operands manipulated by this- conditional ~~branching~~branch instruction, this condition being implemented by the test 513 marked:

$I_{i+1} = IB_{condit.}$

with mOp > 0

the transformation process, ~~in~~on a positive response to the abovementioned test 513, consists of inserting, at a ~~stage~~step 514 before this detected instruction of rank i, a ~~permutation~~swap instruction marked swap_x at the top of the stack of stack variables of the mOp operands of the detected instruction of rank i and the n following values. This ~~permutation~~swap operation makes it possible to collect at the top of the stack of stack variables the n values to be ~~backed up~~stored

in the set of new registers ~~r~~r_1 to r_n. ~~Stage~~Step 514 is followed by a ~~stage~~step 515 consisting in inserting, before the instruction of rank i, a set of ~~backup~~store instructions ~~store~~ to ~~back-up~~save to the set of new registers ~~r~~r_1 to r_n. The abovementioned insertion ~~stage~~step 515 is itself followed by a ~~stage~~step 516 ~~of~~for insertion, after the detected instruction of rank i, of a set of load instructions ~~load~~ to load from the set of new registers ~~r~~r_1 to r_n. The set of corresponding insertion operations is shown in ~~table~~array 12 in the annex.

For reasons of completeness and with reference to ~~fig~~Fig. 5a, it is indicated that, on a negative response to test 504, the continuation of the transformation process is implemented by a context continuation ~~stage~~step, continue 503, that the negative response to tests 505, 508, 511 and 513 is itself followed by a continuation of the transformation process via a context continuation ~~stage~~step, continue 503, and that the same applies to the continuation of operations after the abovementioned redirection ~~stages~~steps 507 and 510 and insertion ~~stages~~steps 512 and 516.

A more detailed description of the method of standardizing and transforming an object code into a standardized object code using typed registers as described in ~~fig~~Fig. 4b will now be given with reference to ~~fig~~Fig. 5b. This ~~implementation mode~~embodiment concerns, more particularly, a nonlimiting, preferred ~~implementation mode of~~ ~~stage~~embodiment of step 601 ~~to reallocate~~for reallocating the registers by detecting the ~~-~~original registers used with different types.

With reference to the abovementioned ~~fig~~Fig. 5b, it is indicated that the abovementioned ~~stage~~step 601 consists in determining, in a ~~stage~~step 603, the lifetime intervals marked ID_j of each register ~~r~~r_j. These lifetime intervals, ~~designated~~called "*live range*" or "*webs*", are defined for a register r as a maximum set of partial traces such that register r is live at all points of these

traces. For a more detailed definition of these concepts, it is useful to refer to the work edited by Steven S. MUCHNICK entitled "Advanced Compiler Design and Implementation", Section 16.3, Morgan KAUFMANN, 1997. ~~Stage~~ Step 603 is designated by the relation:

$$ID_j \leftrightarrow \underline{\leftarrow\rightarrow} r_j$$

~~as claimed in~~according to which a corresponding lifetime interval ID_j is associated with each register r_j.

The abovementioned ~~stage~~step 603 is followed by a ~~stage~~step 604 consisting in determining, at ~~stage~~step 604, the main data type, marked tp_j, of each lifetime interval ID_j. The main type of a lifetime interval ID_j, for a register r_j, is defined by the upper bound of the data types stored in this register r_j by the ~~backup~~store instructions ~~store~~ belonging to the abovementioned lifetime interval.

~~Stage~~Step 604 is itself followed by a ~~stage~~step 605 consisting in establishing an interference graph between the lifetime intervals as defined above at ~~stages~~steps 603 and 604, this interference graph consisting of a non-oriented graph of which each peak consists of a lifetime interval, and of which the arcs, marked $a_{j1, j2}$ on ~~fig~~Fig. 5b, between two peaks ~~ID_{j1}~~ID_{j1} and ID_{j2}, exist if a peak contains a ~~backup~~store instruction addressed to the register of the other peak or vice versa. In ~~fig~~Fig. 5b, the construction of the interference graph is shown symbolically, it being possible to implement this construction on the basis of calculation techniques ~~which are~~ known to those skilled in the art. For a more detailed description of the construction of this type of graph, it is useful to refer to the work published by Alfred V. AHO, Ravi SETHI and Jeffrey D. ULLMAN entitled "Compilers: principles, techniques, and tools", Addison-Wesley 1986, Section 9.7.

Following ~~stage~~step 605, the standardization method as shown in fig. 5b consists in translating, in a ~~stage~~step 606, the uniqueness of a data type ~~which is~~ allocated to each register r_j in the interference graph, by adding arcs between all pairs of peaks of the interference graph while two peaks of a pair of peaks do not have the same associated main data type. It is understood that the translation of the uniqueness character of ~~uniqueness of~~ a data type which is allocated to each register obviously corresponds to ~~translating and taking into account~~the translation and recognition criterion C4 in the interference graph, this criterion being mentioned previously in the description. The abovementioned ~~stage~~step 606 is then followed by a ~~stage~~step 607 in which an instantiation of the interference graph is carried out, this instantiation being more commonly ~~designated~~known as the ~~painting stage~~coloring step of the interference graph ~~as claimed in~~according to the usual techniques. During ~~stage~~step 607, the transformation process assigns to each lifetime interval ID_{jk} a register number rk, in such a way that two adjacent intervals in the interference graph receive different register numbers.

This operation can be implemented on the basis of any suitable process. As a nonlimiting example, it is indicated that a ~~preferred~~ process can consist:

a) in choosing a peak of minimum degree in the interference graph, minimum degree being defined as a minimum number of adjacent peaks, and ~~withdrawing~~removing it from the graph. This ~~stage~~step can be repeated until the graph is empty.

b) Each previously ~~withdrawn~~removed peak is reintroduced into the interference graph in the ~~inverse~~reverse order of their ~~withdrawal~~removal, the last to be removed being the first to be reintroduced, and successively in the ~~inverse~~reverse order of the order of ~~withdrawal.~~ removal. Thus the smallest register number

which is different from the numbers assigned to all the adjacent peaks can be assigned to each reintroduced peak.

Finally, by ~~stage~~step 602, shown in ~~fig~~Fig. 4b, the transformation and reallocation process rewrites the register access instructions ~~to the registers~~ in the code of the ~~subprograms~~subroutine of the relevant applet. Access to a given register in the corresponding lifetime interval is replaced by access to a different register, the number of which has been assigned during the instantiation phase, also designated the ~~painting~~coloring phase.

A more detailed description of an ~~on-board~~embedded data-processing system~~, making it possible to implement~~ for implementing the management ~~protocol~~process and verification process of a program fragment or applet ~~as claimed~~ in accordance with the subject of the present invention, and of a development system of an applet, will now be given with reference to ~~fig~~Fig. 6.

Regarding the corresponding ~~on-board~~embedded system with reference 10, it is recalled that this ~~on-board~~embedded system is a reprogrammable-type system, including the essential components as shown in ~~fig~~Fig. 1b. The abovementioned ~~on-board~~embedded system is considered to be interconnected to a terminal by a serial link, the terminal itself being linked, for instance via a local area network, if appropriate a ~~remote~~wide area network, to an applet development computer with reference 20. On the ~~on-board~~embedded system ~~10 runs~~10, a main program runs which reads and executes the commands ~~which~~sent by the terminal ~~sends on~~over the serial link. Additionally, the standard commands for a microprocessor card, such as for instance the standard commands of the ISO 7816 protocol, can be implemented, ~~and~~ the main program ~~recognizes~~recognizing two additional commands, one for ~~remote loading~~

~~of~~downloading an applet, and the other for selecting an applet which has previously been loaded onto the microprocessor card.

In ~~conformity~~accordance with the subject of the present invention, the structure of the main program is implemented in such a way as to include at least one program module for management and verification of a downloaded program fragment~~, following~~ according to the ~~protocol~~process for managing a downloaded program fragment as described above in the description with reference to ~~fig~~Fig. 2.

Additionally, the program module also includes a ~~subprogram~~subroutine module to verify a downloaded program fragment~~, following~~ according to the verification method as described above in the description with reference to ~~figs~~Figs. 3a to 3j.

For this purpose, the structure of the memories, in particular the non-writable permanent memory ROM, is modified in such a way as to include in particular, ~~apart from~~in addition to the main program, a ~~protocol~~process management and verification module ~~17,~~17 and a virtual machine 16 for interpreting the software code, as mentioned above. Finally, regarding the nonvolatile rewritable memory of EEPROM type, this advantageously includes a directory of applets, marked 18, ~~making it possible to implement~~for implementing the management ~~protocol~~process and the verification process which are the subjects of the present invention.

With reference to the same ~~fig~~Fig. 6, it is indicated that the applet development system conforming to the subject of the present invention, in fact ~~making it possible to transform a traditional~~for transforming a conventional object code as mentioned above in the description, and satisfying criteria C1+C2+C~~13~~'3+C'4 in the framework of the Java environment, into a standardized object code for the same program fragment, includes, associated with a ~~traditional~~conventional Java compiler 21, a code transformation module, marked 22, which

proceeds to transform code into standardized code ~~as claimed in~~according to the first and second ~~implementation modes~~embodiments described above in the description with reference to ~~figs~~Figs. 4a, 4b and 5a, 5b. It is in fact understood that, on the one hand, standardization of the original object code into a standardized object code with ~~branching~~branch instruction with empty stack and into a standardized code using typed registers, on the other hand, as mentioned previously in the description, makes it possible to satisfy verification criteria C3 and C4~~, which are~~4 imposed by the verification method which is the subject of the present invention.

The code transformation module 22 is followed by a ~~JavaCard transformer~~JAVACARD converter 23, which makes it possible to ~~ensure transmission by a remote~~transmit via a wide area or local area network to the terminal and, via the serial link, to the microprocessor card 10. Thus the applet development system 20 shown in fig. 6 ~~makes it possible~~is used to transform the compiled class files produced by the Java compiler 21 from the ~~Java~~JAVA source codes of the applet into class files which are equivalent, but which observe the additional constraints C3, C4 ~~which are~~ imposed by the management ~~protocol~~process and the verification module 17 embedded ~~on board~~ the microprocessor card 10. These transformed class files are ~~transformed~~converted into ~~a downloadable~~an applet ~~on~~which can be downloaded to the card by the standard ~~JavaCard transformer~~JAVACARD converter 23.

Various particularly noteworthy components of the set of ~~protocol~~process components, methods and systems which are the subjects of the present invention will now be given for information only.

Compared to the verification processes of the prior art as mentioned in the introduction to the description, the verification method which is the subject of the present invention appears noteworthy in that it concentrates the verification effort on the typing properties of the operands

which are essential to the security of execution of each applet, i.e. observing the type constraints associated with each instruction and absence of stack overflow. Other verifications do not appear to be essential in terms of security, in particular verification that the code correctly initializes every register before reading it for the first time. On the contrary, the verification method which is the subject of the present invention operates by initializing to zero all the registers from the virtual machine when the method is initialized, to guarantee that reading a non-initialized register cannot compromise the security of the card.

Additionally, the ~~demand~~requirement imposed by the verification method which is the subject of the present invention, ~~as claimed in~~according to which the stack must be empty at each ~~branching~~branch or ~~branching~~branch target instruction, ~~guarantees~~ensures that the stack is in the same state, empty, after execution of the ~~branching~~branch and before execution of the instruction to which the program has branched. This ~~mode of operation guarantees~~procedure ensures that the stack is in a consistent state, whatever the execution ~~route~~path which is followed through the code of the relevant ~~subprogram~~subroutine or applet. The consistency of the stack is thus guaranteed even in the presence of a ~~branching~~branch or ~~branching~~branch target. Contrary to the methods and systems of the prior art, in which it is necessary to ~~conserve~~retain in random-access memory the type of the stack at each ~~branching~~branch target, which necessitates a quantity of random-access memory proportional to TpxNb, the product of the maximum size of execution stack ~~which is~~ used and the number of ~~branching~~branch targets in the code, the verification method which is the subject of the present invention only needs the type of the execution stack at the time of the instruction during verification, and it does not keep in memory the type of this stack at other points of the code. Consequently, the method which is the subject of the invention

is satisfied with a quantity of random-access memory proportional to Tp but independent of Nb, and consequently of the length of the code of the ~~subprogram~~subroutine or applet.

The requirement ~~as claimed in~~according to criterion C4, ~~as claimed in~~according to which a given register must be used with one and the same type throughout the code of a ~~subprogram, guarantees~~subroutine, ensures that the abovementioned code does not use a register in an inconsistent way, e.g. by writing a short integer to it at one point of the program and rereading it as an object reference at another point of the program.

In the verification processes ~~which are~~ described in the prior art, in particular in the previously mentioned Java specification entitled "The Java Virtual Machine Specification", edited by Tim LINDHOLM and Frank YELLIN, to guarantee the consistency of the abovementioned uses through the ~~branching~~branch instructions, it is necessary to keep in random-access memory a copy of the ~~table of~~ register ~~types~~type array at each ~~branching~~branch target. This operation necessitates a quantity of random-access memory proportional to $T_r x N_b$, where T_r ~~designates~~denotes the number of registers used by the ~~subprogram~~subroutine and N_b the number of ~~branching~~branch targets in the code of this ~~subprogram~~subroutine.

On the contrary, the verification process which is the subject of the present invention operates on a global ~~table of~~ register ~~types~~type array without keeping a copy at different points of the code in random-access memory. Consequently, the random-access memory ~~which is~~ required to implement the verification process is proportional to T_r but independent of N_b, and consequently of the length of the code of the relevant ~~subprogram~~subroutine.

The constraint ~~as claimed in~~according to which a given register is used with the same type at all points, i.e. at every instruction of the relevant code, simplifies appreciably and significantly the verification of ~~subprograms.~~ subroutines. On the contrary, in the verification

processes of the prior art, in the absence of such a constraint, the verification process must establish that the ~~subprograms~~subroutines observe a strict stack discipline, and must verify the body of the ~~subprograms~~subroutines polymorphously regarding the type of certain registers.

In conclusion, the verification process which is the subject of the present invention, compared to the techniques of the prior art, makes it possible, on the one hand, to reduce the size of the code of the program ~~code which makes it possible to carry~~for carrying out the verification method, and on the other hand, to reduce the consumption of random-access memory during the verification operations, the degree of complexity being of the form $O(T_p+P_r)$ in the case of the verification process which is the subject of the present invention, instead of $(\theta O(T_p+T_r)\ xN_b)$ for the verification process of the prior art, while however offering the same guarantees ~~about~~regarding the security of execution of the verified code.

Finally, the process of transforming original ~~traditional~~conventional code into standardized code is implemented by localized transformation of the code without transmitting additional information to the verifier component, i.e. the microprocessor card or ~~on-board~~embedded data-processing system.

Regarding the method of reallocating registers as described in figs. 4b and 5b, this method differs from the known methods of the prior art, as described in particular in US Patents 4,571,678 and 5,249,295, by the fact that:

- the register reallocation ensures that the same register cannot be assigned to two intervals with different main types, which thus ~~guarantees~~ensures that a given register is used with the same type throughout the code; and
- the existing register allocation algorithms, which are described in the abovementioned documents, assume a fixed number of registers, and attempt to minimize the transfers, called

"*spills*", between registers and stack, whereas reallocation of registers ~~as claimed~~ in accordance with the subject of the present invention operates in a framework where the total number of registers is variable, as a consequence of which there is no purpose in carrying out transfers between registers and stacks when a process of minimizing the total number of registers is ~~carried out.~~ implemented.

The ~~protocol~~process for managing a program fragment downloaded ~~onto~~to an ~~on-board~~embedded system, and the methods of verifying this downloaded program fragment and respectively of transforming this object code of a downloaded program fragment ~~respectively~~, which are the subjects of the present invention, can of course be implemented ~~in~~by software.

Therefore, the present invention also concerns a computer program product which can be loaded directly into the internal memory of a reprogrammable ~~on-board~~embedded system, this ~~on-board~~embedded system making it possible to download a program fragment consisting of an object code, a series of instructions, executable by the microprocessor of the ~~on-board~~embedded system by ~~way~~means of a virtual machine ~~equipped~~provided with an execution stack and with local registers or variables manipulated ~~via~~by these instructions ~~50~~so that this object code can be interpreted. The corresponding computer program product includes portions of object code to execute the ~~protocol~~process for managing a program fragment downloaded ~~onto~~to this ~~on-board~~embedded system, as shown in ~~figs~~Figs. 2 and 6 described above in the description, when this ~~on-board~~embedded system is interconnected ~~to~~with a terminal and this program is executed by the microprocessor of this ~~on-board~~embedded system by ~~way~~means of the virtual machine.

The invention also concerns a computer program product which can be loaded directly into the internal memory of a reprogrammable ~~on-board~~embedded system, such as a microprocessor card with a rewritable memory, as shown with reference to ~~fig~~Fig. 6. This

computer program product includes portions of object code to execute the ~~stages of~~steps for verifying a program fragment downloaded ~~onto~~to this ~~on-board~~embedded system, as shown and described above in the description, with reference to ~~figs~~Figs. 3a to 3j. This verification is executed when this ~~on-board~~embedded system is interconnected ~~to~~with a terminal and this program is executed by the microprocessor of this ~~on-board~~embedded system ~~via~~by means of the virtual machine.

The invention also concerns a computer program product; this computer program product includes portions of object code to execute the ~~stages~~steps of the method of transforming the object code of a program fragment into standardized object code for this same program fragment, as shown in ~~figs~~Figs. 4a, 4b, 5a, 5b and 6, and described above in the description.

The present invention also concerns a computer program product which is ~~recorded~~stored on a medium which can be used in a reprogrammable ~~on-board~~embedded system, e.g. a microprocessor ~~equipped~~provided with a rewritable memory, this ~~on-board~~embedded system ~~making it possible~~being used to download a program fragment consisting of an object code executable by this microprocessor, by ~~way~~means of a virtual machine ~~equipped~~provided with an execution stack and local variables or registers manipulated ~~via~~by these instructions, to ~~allow~~enable interpretation of this object code. The abovementioned computer program product includes, at least, a module of programs which can be read by the microprocessor of the ~~on-board~~embedded system ~~via~~by means of the virtual machine, to ~~command~~control the execution of a procedure for managing the downloading of a downloaded program fragment, as shown in ~~fig~~Fig. 2 and described above in the description, a module of programs which can be read by the microprocessor ~~via~~by means of the virtual machine, to ~~command~~control the execution of a procedure for verifying, instruction by instruction, the object code which makes up this program

fragment, as shown and described in relation to ~~figs~~Figs. 3a to 3j in the description above, and a module of programs which can be read by the microprocessor of this ~~on-board~~embedded system ~~via~~by means of the virtual machine, to ~~command~~control the execution of a downloaded program fragment following or in the absence of a transformation of the object code of this program fragment into a standardized object code for this same program fragment, as shown in ~~fig~~Fig. 2.

The abovementioned computer program product also includes a module of programs which can be read by the microprocessor ~~via~~by means of the virtual machine, to ~~command~~control the inhibition of execution, on the ~~on-board~~embedded system, of the program fragment in the case of an unsuccessful verification procedure ~~of~~on the abovementioned program fragment, as shown and described above in the description with reference to ~~fig~~Fig. 2.

What is claimed is:

ANNEXES

~~# 4513354_v1~~

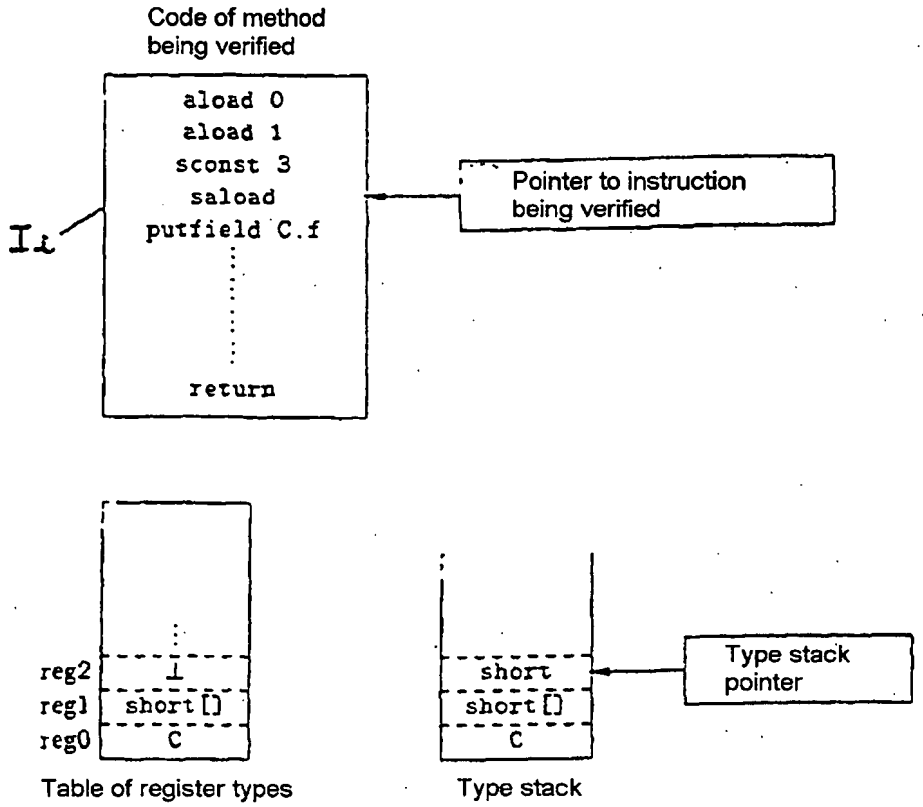

Code of method
being verified
aload 0
aload 1
sconst 3
saload
putfield C.f
return
I_i
Pointer to instruction
being verified
reg2
reg1
reg0
⊥
short[]
C
Table of register types
short
short[]
C
Type stack
Type stack
pointer

TABLE 2

Pseudo-code of verifier module

PSEUDO-CODE OF VERIFIER MODULE

Global variables used:

T_r	number of registers declared by current method
T_p	maximum size of stack declared by current method
$tr[T_r]$	table of register types (402 in fig. 4)
$tp[T_p]$	stack type (403 in fig. 4)
pp	stack pointer (404 in fig. 4)
chg	flag indicating whether tr has changed

Initialize $pp \leftarrow 0$

Initialize $tp[0]$ ... $tp[n-1]$ from types of n arguments of method

Initialize $tp[n]$ ... $tp[T_r-1]$ to $\perp$

Initialize chg to true

While chg is true:

    Reset chg to false

    Position on first instruction of method

    While end of method is not reached:

        If current instruction is target of a branching instruction:

            If $pp \neq 0$, verification fails

        If current instruction is target of a subroutine call:

            If previous instruction continues in sequence, failure

            Take $tp[0] \leftarrow$ etaddr and $pp \leftarrow 1$

        If current instruction is an exception handler of class C:

            If previous instruction continues in sequence, failure

            Do $tp[0] \leftarrow C$ and $pp \leftarrow 1$

        If current instruction is a target of different kinds:

            Verification fails

        Determine types a_1 ... a_n of arguments of instruction

        If $pp < n$, failure (stack overflow)

        For $i = 1, ..., n$:

            If $tp[pp-n-i-1]$ is not subtype of a_i, failure

        Do $pp \leftarrow pp-n$

        Determine types $r_1, ..., r_m$ of results of instruction

        If $pp+m \geq T_p$, failure (stack overflow)

        For $i = 1, ..., m$, do $tp[pp+i-1] \; ? \; r_i$

        Do $pp \leftarrow pp+m$

        If current instruction is a write to a register r:

Determine type *t* of value written to record

Do *tr*[*r*]←*lower bound* (*t*, *tr*[*r*])

If *tr*[*r*] has changed, do *chg*←true

If current instruction is a branching:

If *pp*≠0, verification failure

Advance to next instruction

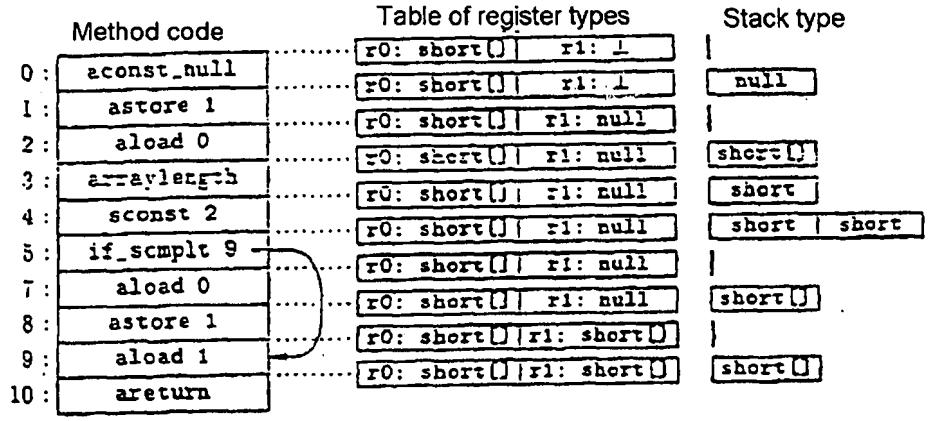
Return verification success code

TABLE T3

```
static short[] meth(short [] table    )
{
    short[] result      = null;
    if (  table length     >= 2) result = table;
     return table
}
```

TABLE T4

First iteration on method code:



Second iteration on method code:

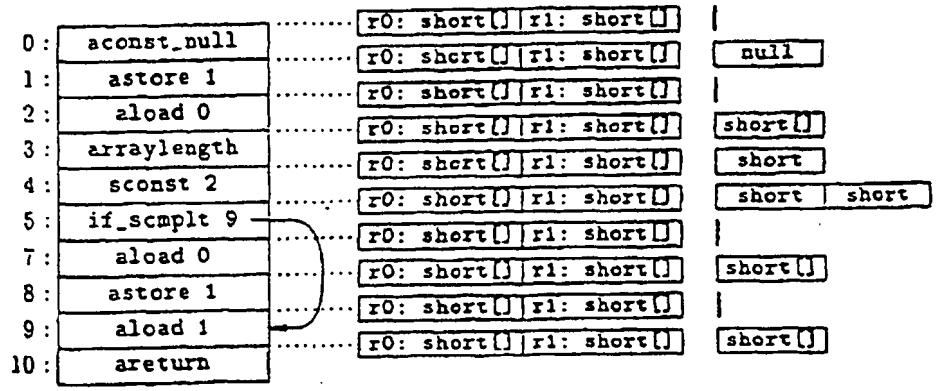
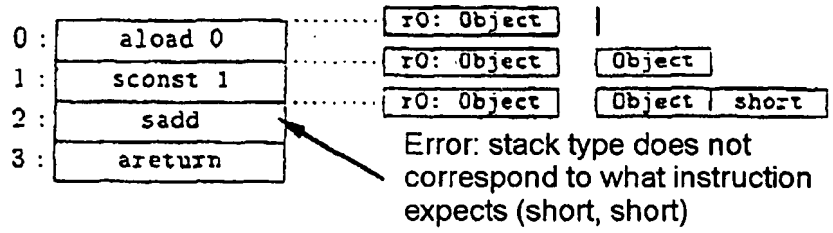
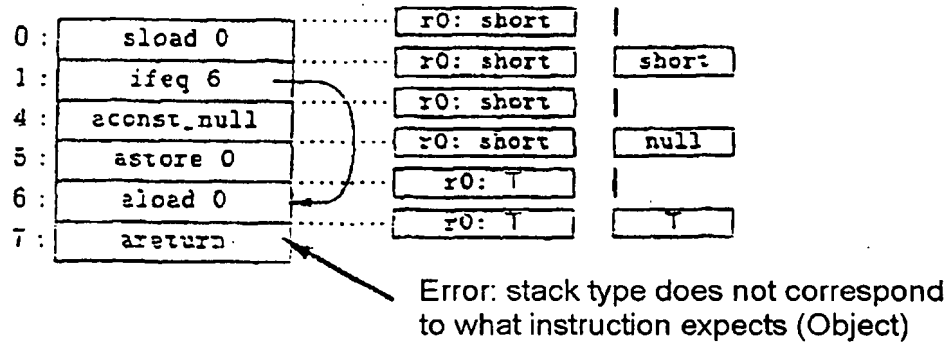


TABLE T5

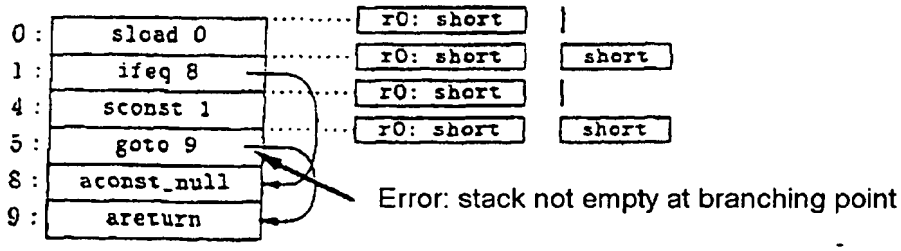
(a) *Violation of type constraints on arguments of an instruction:*



(b) *Inconsistent use of a register:*



(c) *Branchings introducing inconsistencies at stack level:*



(d) *Stack overflow within a loop:*

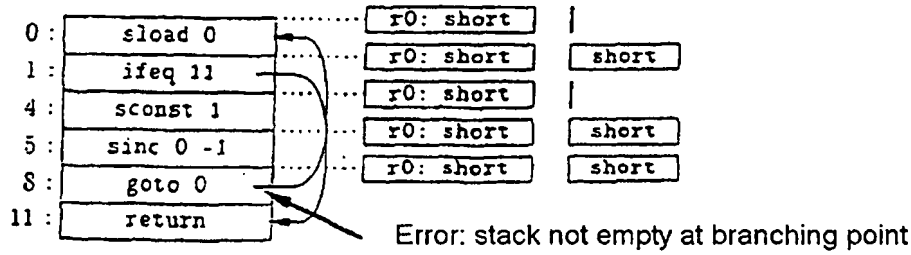
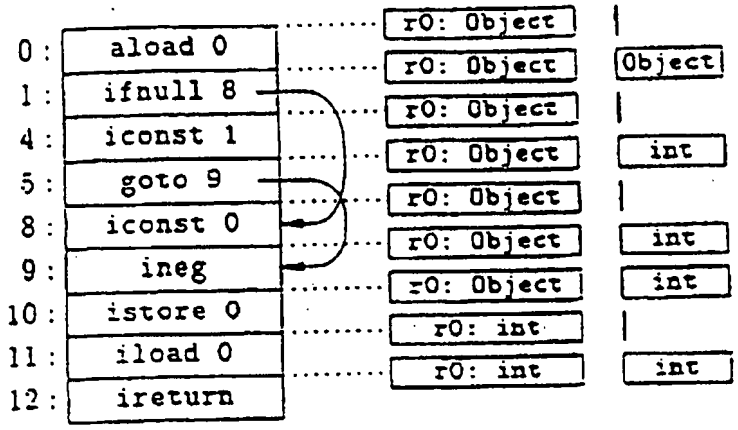


TABLE T6

(a) *Initial code of method, annotated by types of registers and of stack:*



5 TABLE T7

(b) *Method code after standardization of stack at branching 5 → 9:*

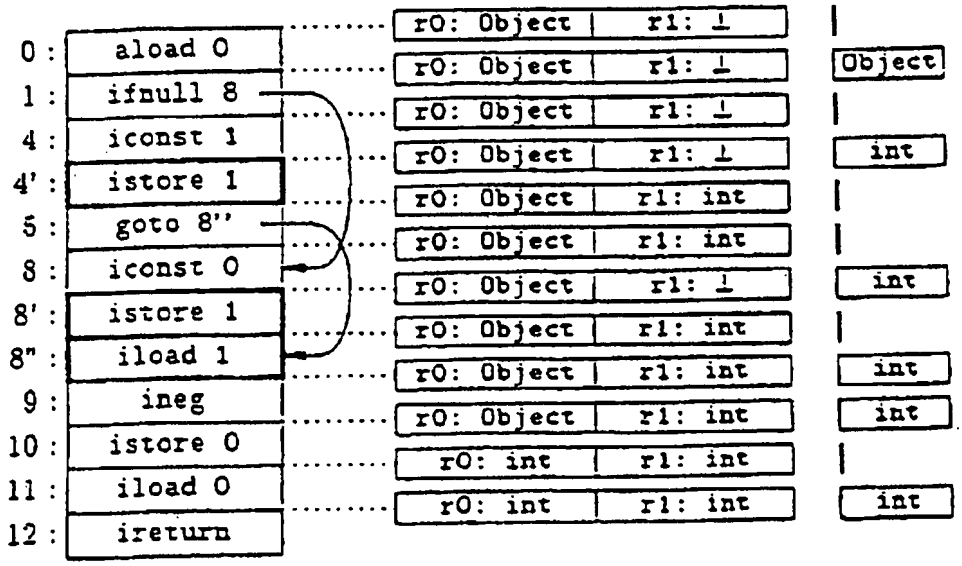
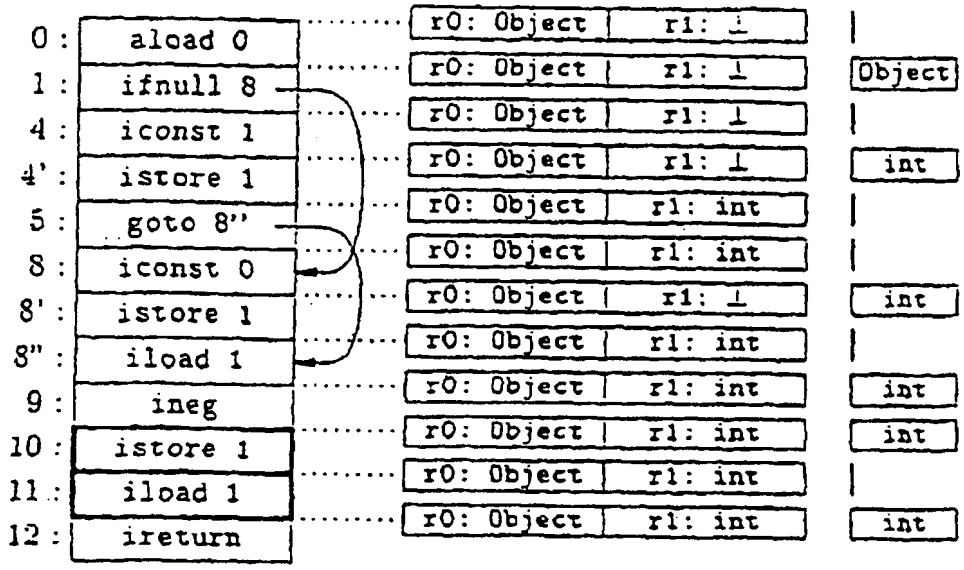


TABLE T8

(c) *Method code after reallocation of registers:*



5 TABLE T9

(a) *Branching target, previous instruction not continuing in sequence:*

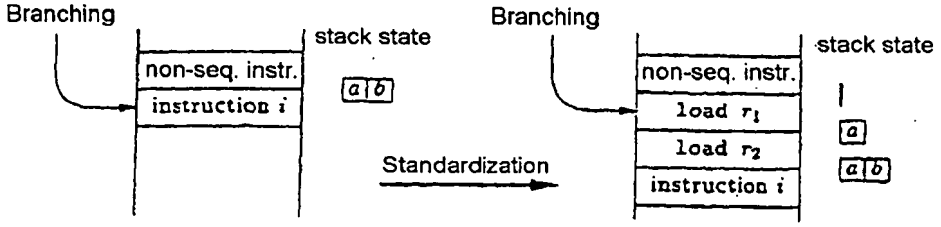


TABLE T10

(b) *Branching target, previous instruction continuing in sequence:*

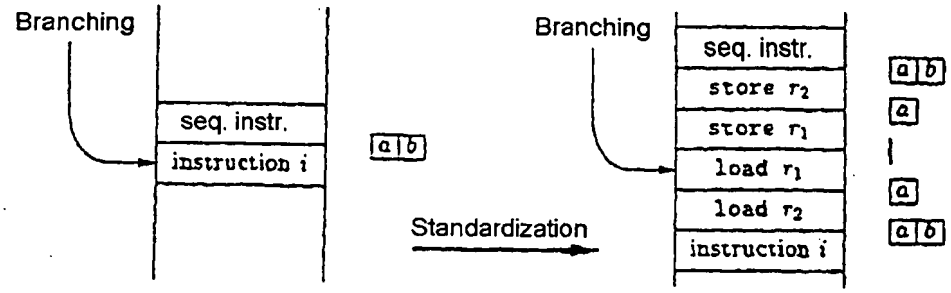


TABLE T11

(c) *Unconditional branching without arguments:*

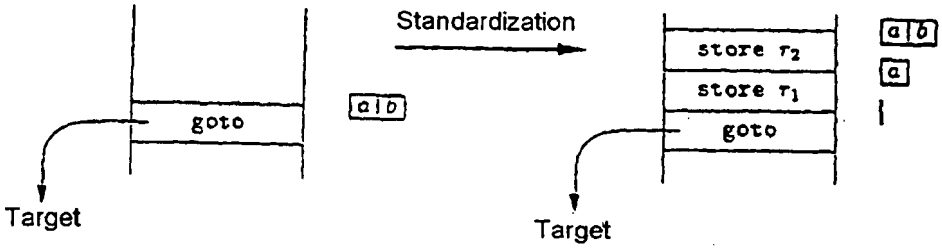


TABLE T12

(c) *Conditional branching with one argument:*